

Part II. *Smoldyn* Code Documentation

Steven Andrews

The main code is separated into three files. `smoldyn.c` contains the `main()` function, some high level functions for running the simulation, and the graphics routines. `smolllib.c` and its header `smolllib.h` contain the structure declarations and low level routines. Functions include ones that allocate and free memory, calculate simulation parameters, set up the simulation, run the simulation, and provide diagnostics. This is the core of the program. Finally, `smolllib2.c` and its header `smolllib2.h` implement commands for the runtime interpreter. It is expected that more commands will be desired on a regular basis, so this library will be appended regularly.

1. Compiling *Smoldyn*

For the most part, *Smoldyn* is written in plain ANSI C. Minor exceptions are that a few very minor and standard updates to strict ANSI are used, such as the characters “//” to indicate a comment. Also, *Smoldyn* uses OpenGL graphics which requires the graphics libraries `gl.c`, `glut.c`, and their header files. These libraries are available on a wide variety of systems including Macintosh, Windows, and Unix. To save images as TIFF files, *Smoldyn* also includes an enormous library of files that were written by Sam Leffler. Most of them are not used at all, but I couldn’t seem to separate the useful from the useless ones. In principle, neither the OpenGL nor the TIFF libraries are needed but they are often very useful.

Proper file linking seems to be a mystical aspect of C programming, so getting all these files to load and link properly can be a major challenge. Following are a few words of advice.

Macintosh, Windows, and Unix (including Linux) all use different line termination characters. Macintosh uses a carriage return, Unix uses a line feed, and Windows uses both. If files aren’t converted properly, this will cause problems. I think that most Macintosh and Windows compilers convert code automatically, but this needs to be checked for your particular compiler. However, Unix does not. There are various ways to convert files in Unix; for example, you can use the “sed” command, or there is a file-conversion option that you can select in the “mule” menu in emacs.

2. Include files, macros, variables, etc.

2.1 Include files

`smoldyn.c, non-OpenGL`

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
```

```
#include "smolib.h"
#include "smolib2.h"
#include "string2.h"
#include "SimCommand.h"
```

smoldyn.c, with OpenGL

```
#include "opengl2.h"
#include <gl.h>
#include <glut.h>
```

smolib.h

```
#include "SimCommand.h"
```

smolib.c

```
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <ctype.h>
#include "Rn.h"
#include "Zn.h"
#include "random.h"
#include "string2.h"
#include "math2.h"
#include "rxnparam.h"
#include "SimCommand.h"
#include "smolib.h"
#include "smolib2.h"
#include "VoidComp.h"
#include "Geometry.h"
#include "opengl2.h"
```

smolib2.h

```
#include "SimCommand.h"
#include "smolib.h"
```

smolib2.c

```
#include <stdio.h>
#include <math.h>
#include <string.h>
#include "Rn.h"
#include "Zn.h"
#include "random.h"
#include "string2.h"
#include "smolib2.h"
#include "opengl2.h"
#include "SimCommand.h"
```

```
#include "opengl2.h"
#include <gl.h>
#include <glut.h>
```

In principle, these are simple and self-explanatory. As noted above though, proper linking and compiling can be challenging.

2.2 Constants and global variables

smoldyn.c

```
#define SMOLDYN_VERSION 1.72
simptr Sim;
int Vb,*Ctr;
time_t Tstt;
```

smolib.c

```
#define RANDTABLEMAX 4095
#define PSMAX 5
double GaussTable[RANDTABLEMAX+1];
```

The notation used is that macros and constants defined with the pre-processor are in all capitals, global variables are preceded by a capital letter, and local variables are in all lower case. Variables that have a meaning have meaningful names, whereas those that are scratch space have generic names that simply indicate the variable types. All of these macros and global variables are global only within their source file, so that they are not accessible to other files.

SMOLDYN_VERSION is the current version number of Smoldyn.

Sim is a global variable for the current simulation structure. This, as well as the other global variables in smoldyn.c are only used when graphics are being shown using OpenGL, because OpenGL does not allow variables to be passed in the normal way between functions.

Vb is a global flag for verbose operation, equal to 1 if yes, 0 if no.

Ctr is a global array of simulation counters, which count the total number of events that occur during a simulation. Index 0 is for zeroth order reactions, 1 is for first order reactions, 2 is for second order reactions within a partition, 3 is for second order reactions between partitions, and 4 is for wall collisions.

Tstt is a global variable for the starting time of simulation execution, allowing the total runtime to be determined.

RANDTABLEMAX is the maximum element number of the random number conversion table, which is used both for allocating the table (with 1 more element) and as a bit mask for random number routines. Note that it needs to be 1 less than an integer power of 2.

PSMAX is the maximum number of panel shapes defined. Currently it is 5 for rectangle, triangle, sphere, cylinder, and hemisphere.

GaussTable is a table to convert uniform random values to normally distributed ones.

2.3 Macros

smolib.c

```
#define CHECK(A) if(!(A)) goto failure
#define CHECKS(A,B) if(!(A)) {strncpy(erstr,B,STRCHAR);goto failure;}
```

CHECK is a useful macro for several routines in which any of several problems may occur, but all problems result in freeing structures and leaving. Program flow goes to the label failure if A is false. Many people would consider both the use of a macro function and the use of a goto statement to be bad programming practice, and especially bad when used together. However, in this case it significantly improves code readability. As usual, partially defined structures should always be kept traversable and in good order so they can be freed at any time. However, there is one subtle situation where CHECK can cause surprising behavior, which must be looked out for:

```
if(test)
    CHECK(a==b);
else {... }
```

WRONG

The problem is that CHECK is a macro for an if() statement, so the else in the above example becomes an else for the CHECK, rather than an else for the if(test) portion, as intended. Instead, this should be coded with braces:

```
if(test) {
    CHECK(a==b); }
else {... }
```

RIGHT

CHECKS is identical to CHECK, except that it also copies the included string to the variable erstr if a failure occurs. This is useful for error reporting. The same comments made above are important here as well.

2.4 Local variables

It has proven useful to use consistent names for local variables for code readability. In places, there are exceptions, but the following table lists the typical uses for most local variables:

variable	type	use
a	double	binding radius for bimolecular reaction

b,b2	int	box address
blist	boxptr*	list of boxes, index is [b]
boxs	boxssptr	pointer to box superstructure
bptr	boxptr	pointer to box
bval	double	unbinding radius for bimolecular reaction
ch	char	generic character
cmd	cmdptr	pointer to a command
cmds	cmdssptr	pointer to the command superstructure
d	int	dimension number
dc1,dc2	double	diffusion coefficients for molecules
dead	moleculeptr*	list of dead molecules, index [m]
difc	double*	list of diffusion coefficients, index [i]
dsum	double	sum of diffusion coefficients
dim	int	dimensionality of space
dt	double	time step
er	int	error code
erstr	char*	error string
flt1,flt2	double	generic double variable
fptr	FILE*	file stream
got	int[]	flag for if parameter is known yet
i	int	molecule identity, reactant number, or generic integer
i1,i2,...	int	molecule identities
indx	int*	dim dimensional index of box position
itct	int	count of number of items read from a string
j	int	number of reaction for certain i
lctr	int	line number counter for reading text file
line	char[]	complete line of text
line2	char*	pointer to unparsed portion of string
live	moleculeptr**	list of live molecules, index [ll][m]
ll	int	index of live list
m,m2,m3	int	index of molecule in list
m1,m2,m3	int*	scratch space matrices of size dimxdim
mlist	moleculeptr*	list of molecules, index is [m]
mols	molssptr	pointer to molecule superstructure
mptr	moleculeptr	pointer to molecule
mptr1,mptr2	moleculeptr	pointer to more molecules
name	char**	names of molecules, index is [i]
nbox	int	number of boxes
ni2o	int	value of nident^order
nident	int	number of molecule identities
nl	int*	number of live molecules in a live list, index [ll]
nm,nm2	char[]	name of molecule, reaction, or surface
nmol	int	number of molecules in list
npnl	int	number of panels for a surface
npts	int	number of points for a surface panel
nprod	int*,int	number of products for reaction [r]

nrxn	int*	number of reactions for [i]
nsrf	int	number of surfaces
o2	int	order of second reaction
optr	int*	pointer to the order of reaction
order	int	order of reaction
p	int	reaction product number
p	int	panel number for surfaces
pgemptr	double*	pointer to probability of geminate recombination
point	double**	list of points that define a panel [p][pt]
prod	moleculeptr**	list of products for reaction [r], index is [p]
pnl	panelptr	pointer to a panel
pnl	panelptr*	list of panels
ps	int	panel shape, 0=rect, 1=triangle, 2=sphere
pshape	char	panel shape, 'r'=rect, 't'=triangle, 's'=sphere
pt	int	index for points, for surfaces
r	int	reaction number
r2	int	reaction number for second reaction
rate	double*	requested rate of reaction [r]
rate2	double*	internal rate parameter of reaction [r]
rate3	double	actual rate of reaction
rev	int	code for reaction reversibility; see findreverserxn
rname	char**	names of reactions, index is [r]
rpar	double,double*	reversible parameter, index is [r]
rpart	char,char*	reversible parameter type, index is [r]
rptra	int*	pointer to reaction number
rxn	rxnptr	pointer to a reaction structure
rxn2	rxnptr	pointer to a second reaction structure
s	int	surface number
side	int*	number of boxes on each side of space, index [d]
sim	simptr	pointer to simulation structure
smpttr	simptr*	pointer to pointer to simulation structure
srf	surfaceptr	pointer to a surface structure
srfss	surfacessptr	pointer to a surface superstructure
step	double	rms step length of molecule or molecules
str1	char[]	generic string
table	int**	table of reaction numbers for [i][j]
topd	int	top of empty molecules in dead list
total	int	total number of reactions in list
v1,v2,v3	int*	scratch space vectors of size dim
w	int	index of wall
wlist	wallptr*	list of walls, index is [w]
word	char[]	first word of a line of text
wptr	wallptr	pointer to wall

3. Structures and core functions – smolib.h, smolib.c

While *Smoldyn* is written in C, it uses an object oriented approach to programming, making the proper maintenance of structures one of the central aspects of the program. The structures and core functions are described below. In general, the basic objects are molecules, walls, surfaces, and virtual boxes, each of which has its own structure. In many cases, these items are grouped together into superstructures, which are basically just a list of fundamental elements, along with some more information that pertains to the whole list. Reactions aren't really objects, but are also among the core data structures. Finally, a simulation structure is a high level structure which contains all the parameters and the current state of the simulation.

An aspect of structures that is important to note, especially if changes are made, is which structures own what elements. For example, a molecule owns its position vector, meaning that that piece of memory was allocated with the molecule and will be freed with the molecule. On the other hand, a molecule does not own a virtual box, but merely points to the one that it is in.

All allocation routines return either a pointer to the structure that was allocated, or NULL if memory wasn't available. Assuming that they succeed, all structure members are initialized, typically to 0 or NULL depending on the member type. All the memory freeing routines are robust in that they don't mind NULL inputs or NULL internal pointers. However, this is only useful and robust if allocation is done in an order that always keeps the structure traversable and keeps pointers set to NULL until they are ready to be initialized.

Both the code and the description below are sorted into categories: molecules, walls, reactions, boxes, surfaces, and the simulation structure. In many cases, functions within each category work with only their respective object. However, be forewarned that the core program is highly integrated so that functions in one category may use objects in another category. While there are a few exceptions, in general, functions in one category do not write to objects in other categories.

3.1 Molecules

Each individual molecule is stored with a `moleculestruct` structure, pointed to by a `moleculeptr`. This contains information about the molecule's position, identity, and other characteristics that are specific to each individual molecule. These molecules are organized using a molecule superstructure, which contains lists of the active molecules, a list of unused molecule storage space called dead molecules, and lots of other information about the molecules in general.

<code>typedef struct moleculestruct {</code>	
<code> long int serno;</code>	serial number
<code> double *pos;</code>	dim dimensional vector for position
<code> double *posx;</code>	dim dimensional vector for old position
<code> int ident;</code>	identity of molecule; 0 is empty
<code> struct boxstruct *box;</code>	pointer to box which molecule is in
<code> struct panelstruct *pnl;</code>	last panel interacted with
<code>}</code>	

char face;	face of last panel interacted with
double *via;	location of last surface interaction
} *moleculeptr;	

moleculestruct is a structure used for each molecule. serno is the unique serial number that each live molecule has; it is updated when the molecule is placed on the live list. pos and posx, both of which are owned by the structure, are always valid positions, although not necessarily within the system volume. posx is the position from the previous time step, used to determine if a molecule crossed a wall or surface. ident should always be between 0 and nident-1, inclusive. A molecule type of 0 is an empty molecule for transfer to the dead list. Except during setup, box should always point to a valid box. If this molecule interacts with a surface during the current time step, pnl points to the most recent panel hit, face is the face of the panel that was hit, and via is the location of that interaction.

typedef struct molsuperstruct {	
double *difc;	diffusion constants for each identity
double *difstep;	rms diffusion step for each identity
double **difm;	diffusion matrix for each identity
double *display;	size of molecule in graphical display
double **color;	RGB color vector for each identity
double **grphdata;	all position data for graphics
int *grphser;	number of rows in columns of grphdata
moleculeptr *live[2];	live molecules in system (0 mobile, 1 fixed)
moleculeptr *dead;	list of dead molecules
int max;	size of each molecule list
int nl[2];	number of molecules in live lists
int topl[2];	index for live lists; above are reborn
int nd;	total number of molecules in dead list
int topd;	index for dead list; above are resurrected
long int serno; } *molssptr;	serial number for next resurrected molec.

molsuperstruct contains and owns information about molecular properties and it also contains and owns three lists of molecules. Diffusion is described with difc, which is a maxident length vector of diffusion constants; difstep is a maxident length vector of the rms displacements on each coordinate during one time step if diffusion is isotropic; and difm is a maxident length list where each element is either a NULL value if diffusion is isotropic or a dimxdim size diffusion matrix (actually the square root of the matrix). display is simply the size of molecules for graphical output (which scales differently for different output styles) and color is the 3-dimensional color vector for each molecule. grphdata and grphser are data space that are not used by the simulation, but are filled in and used for the graphical output.

The molecule lists are separated into two parts. The first set is the live list, which are those molecules that are actually in the system; the others are in the dead list, are empty molecules, and have no influence on the system. The two parts of the live list are the mobile molecules (live[0]) and the fixed molecules (live[1]). They are differentiated solely by whether their diffusion constants are zero and are separated into

two lists to speed up bimolecular reaction routines. All lists have size `max` (the total number of molecules allocated), which means that any list can contain all molecules, if needed. If more molecules are needed in the system than the total number allocated, the program sends an error message and ends; in the future, it may be possible to dynamically create larger lists. Upon initialization, all molecules are created as empty molecules in the dead list and both live lists are full of NULLs, whereas during program execution, all lists are typically partially full. After sorting, each live list, `ll`, has active molecules from element 0 to element `nl[ll]-1`, inclusive, and has undefined contents from `nl[ll]` to `max-1`. Similarly, the dead list is filled with empty molecules from 0 to `nd-1`, and has undefined contents from `nd` to `max-1`; in this case, `topd` equals `nd`.

Functions other than `molsort`, such as chemical reactions, are allowed to kill live molecules or resurrect dead ones but they should not move molecules or change the list indicies except for `topd`. To kill a molecule from one of the live lists, just set the molecule identity to 0. To resurrect a dead molecule, decrement `topd` by one (making sure it wasn't equal to 0) and set the identity of the now `topd` molecule in the dead list as desired. Also, set the `box` element of the molecule to point to the proper box, but do not add the molecule to that box's molecule list. It is now in the resurrected list, which is the top of the dead list between `topd` and `nd-1`, inclusive. Routines should be written so that these mis-sorted molecules do not cause problems. To sort them, call `molsort`, which moves the empty molecules in the live lists to the dead list, moves the resurrected ones to the top of the proper live list, compacts the live lists while maintaining the molecule order, and finally identifies the newly reborn molecules in the live lists by setting `topl[ll]`; the reborn molecules extend from `topl[ll]` to `nl[ll]`.

Example of the lists:

index	live[0]		live[1]		dead	
8	max	?	max	?	max	?
7		-		-		-
6		-		-		-
5		-		-		-
4		-		-		-
3	nl[0]	-		-	nd	-
2		2	topl[1],nl[1]	-	topd	1
1	topl[0]	1		0		0
0		0		3		0

Here, each list has `max=8`, and so is indexed with `m` from 0 to 7. A '?' is memory that is not part of that which was allocated, a '-' is a NULL value, a '0' is an empty molecule, and other numbers are other identities ('1' and '2' are mobile, whereas '3' is immobile). The '0's in the the two live lists are to be transferred to the dead list during the next sort, while the '1' in the dead list has been resurrected and is to be moved to mobile live list. Based on the `topl[0]` index, it can be seen that the '1' and '2' in the mobile live list were just put there during the last sorting, and so are reborn molecules.

```
moleculeptr molalloc(int dim)
```

`molalloc` allocates and initializes a new `moleculestruct`. The `box` and `diffusion` matrix members are returned as `NULL`.

`void molfree(moleculeptr mptr)`
`molfree` frees the space allocated for a `moleculestruct`, as well as its two position vectors.

`molssptr molssalloc(int dim,int max,int maxident);`
`molssalloc` allocates and initializes a molecule superstructure with `max` molecule spaces in each of the three lists. `max` must be at least 1. The dead list is filled with empty molecules. The molecule boxes are left as `NULLs`, and need to be set. Book keeping elements for the lists are set to their initial values.

`void molssfree(molssptr mols,int maxident);`
`molssfree` frees both a superstructure of molecules and all the molecules in all its lists.

`void molssoutput(simptr sim);`
`molssoutput` prints all the parameters in a molecule superstructure. While it should not be needed, hopefully, this routine looks for and prints out information on molecules that are not sorted correctly in the live and dead lists. It also prints out information about each molecule, including diffusion constants, rms step lengths, colors, and display sizes.

`void setdiffusion(simptr sim);`
This sets up the diffusion coefficient related aspects of the molecule superstructure. If `difm` is initialized but not `difc`, then it creates `difc`. It also assigns `difstep`.

`int molsort(molssptr mols,int difsort);`
`molsort` updates the live and dead lists of both a molecule superstructure and the relevant boxes after a reaction or other changes. If `difsort` is 0, it assumes that live molecules are in the correct live list, otherwise, it sorts this too (the only way it's likely to need sorting is if a command changes a molecule's identity). First it deals with diffusion sorting, if requested, by moving missorted molecules to the resurrected list. Afterwards, it moves the resurrected molecules off the top of the dead list (those numbered between `top` and `nd-1`) to the live list. Then it moves the expired molecules from the live list to the dead list. Finally it compacts the live list. Molecule ordering in lists is preserved. Molecules don't have to be in the correct boxes, but it is assumed that each molecule that has been killed is in a box: each molecule's box element must point to a box, and those boxes' molecule lists must list the respective molecules. Resurrected molecules need to have the proper box listed in the molecule structure, but should not be listed in the box list; this listing is taken care of here. The routine returns 0 for normal operation and 1 if memory could not be allocated when a box was being expanded.

`void diffuse(simptr sim);`

diffuse does the diffusion for all molecules in the `live[0]` list (mobile), over one time step. Walls are ignored and molecules are not reassigned to the boxes. If there is a diffusion matrix, it is used for anisotropic diffusion; otherwise isotropic diffusion is done, using the `difstep` parameter. Assuming molecules were assigned to the proper boxes initially, the new `posx` molecule element (prior position) should be inside the box that is listed, and in which the molecule is still listed.

3.2 Walls

The simulation volume is defined by its bounding walls. If no other surfaces are defined, these walls can be reflecting, periodic, absorbing, or transparent. Because walls can be transparent, molecules can leave the simulation volume. However, this can be a bad idea because the virtual boxes are defined to exactly fill the volume within the walls, so molecules or surfaces outside of the simulation volume can lead to very slow simulations. Also, the graphics are designed for the simulation volume within the walls. If surfaces are defined, then walls, regardless of how they are set up, are simulated as though they are transparent.

Walls are quite simple, defined with only a simple structure and no superstructure. A simulation always has $2 \cdot \text{dim}$ walls.

```
typedef struct wallstruct {
    int wdim;           dimension number of perpendicular to wall
    int side;           low side of space (0) or high side (1)
    double pos;         position of wall along dim axis
    char type;          properties of wall
    struct wallstruct *opp; } *wallptr;  pointer to opposite wall
```

`wallstruct` (declared in `smollib.h`) is a structure used for each wall. The type may be one of four characters, representing the four possible boundary conditions.

<u>type</u>	<u>boundary</u>
r	reflecting
p	periodic
a	absorbing
t	transparent

Pointers to the opposite walls are used for wrap-around diffusion, but are simply references. There is no superstructure of walls, but, instead a list of walls is used. Walls need to be in a particular order: walls numbered 0 and 1 are the low and high position walls for the 0 coordinate, the next pair are for the 1 coordinate, and so on up to the $2 \cdot \text{dim} - 1$ wall. These walls are designed to be bounds of simulated space, and are not configured well to act as membranes.

```
wallptr walllalloc(void);
```

`walllalloc` allocates and initializes a new wall. The pointer to the opposite wall needs to be set.

```
void wallfree(wallptr wptr);
    wallfree frees a wall.

wallptr *wallsalloc(int dim);
    wallsalloc allocates an array of pointers to 2*dim walls, allocates each of the walls,
    and sets them to default conditions (reflecting walls at 0 and 1 on each coordinate)
    with correct pointers in each opp member.

void wallsfree(wallptr *wlist,int dim);
    wallsfree frees an array of 2*dim walls, including the walls.

void walloutput(int dim,wallptr *wlist);
    walloutput prints the wall structure information, including wall dimensions,
    positions, and types, as well as the total simulation volume.

int checkwalls(simptr sim);
    checkwalls does the reflection, wrap-around, or absorption of molecules at walls by
    checking the current position, relative to the wall positions (as well as a past
    position for absorbing walls). It does not reassign the molecules to boxes or sort the
    live and dead ones. Ideally, molecules should be assigned to the box where the
    molecule's posx location is, although results are almost certainly identical if they
    are assigned to the box for the pos location. It returns the number of wall collisions
    that were detected and processed during that time step.
```

3.3 Reactions

Despite the fact that reactions are stored in only one reasonably small structure, they are still complicated.

typedef struct rxnstruct {	
int order;	order of reactions listed: 0, 1, or 2
int *nrxn;	number of reactions for each set of reactants
int **table;	lookup list of reaction numbers
int lists;	live lists that have reactions
int total;	total number of reactions listed
char **rname;	names of reactions
double *rate;	list of requested reaction rates
double *rate2;	reaction rates modified for computation
double *rpar;	parameter for reaction of products
char *rpart;	type of parameter in rpar
int *nprod;	number of products for each reaction
moleculeptr **prod; } *rxnptr;	templates of products for each reaction

rxnstruct (declared in smollib.h) is a structure used for a complete set of zeroth order, unimolecular, bimolecular reactions, or higher order reactions. All

components of all lists in the structure are owned by the structure. While these structures are complicated, they are also quite versatile and fast to use. The `table` member is a lookup table of all possible reactant combinations, returning an index value of the reaction, if one occurs, called a reaction number. The reaction parameters, such as the reaction names, rates, and product list are then listed sequentially in order of reaction numbers. The dimensionality of the lookup table is the reaction order. If order is 0, then there are no reactants to worry about; `nrxn` and `table` are allocated and initialized so that `nrxn[0]=0` and `table[0]=NULL`, and there are no higher indices allowed. If order is 1, then `nrxn[i]` is the number of unimolecular reactions that molecules of type `i` can undergo and `table[i]` is a list of `nrxn[i]` reaction numbers. For example, `table[i][j]` is the reaction number of the `j`'th unimolecular reaction for molecule `i`. Clearly, empty molecules are included in these lists, accessed with `nrxn[0]` and `table[0]`, where the former should always equal 0 and the latter should always be `NULL`. If order is 2, the same scheme is followed, although now `i` is an index for a two dimensional array. For example, the number of bimolecular reactions possible between molecules `i1` and `i2` is found by first defining `i=maxident*i1+i2` with the result of `nrxn[i]` possible reactions. For all reaction orders, `nrxn` and the first index of `table` extend from 0 to `maxidentorder`. `lists` is a parameter used to prevent having to scan molecule lists for reactions that don't exist. For zeroth order reactions, `lists` is set to 0. For first order reactions, `lists` is 0 if no molecules have any reactions, 1 if only mobile molecules have reactions, 2 if only immobile molecules have reactions, and 3 for both. For second order reactions, `lists` is 0 if no molecules have any reactions, 1 for only mobile-mobile, 2 for only mobile-immobile, 3 for both.

`total` is the total number of reactions listed in the structure. `rate`, `rate2`, `rpar`, `rpart`, `nprod`, and the first indices of `rname` and `prod` are all allocated to have `total` elements. The reaction rates are given in `rate`, although these are generally bulk reaction rates rather than microscopic molecular parameters. `rate2` is the rate information used by the simulation routines: if order is 0, `rate2` is the average number of molecules produced per time step; if order is 1, `rate2` is the probability that a molecule reacts during one time step; if order is 2, `rate2` is the squared collision distance between the relevant pair of reactant molecules. Note that the value of `rate` is independent of the time step, whereas the value of `rate2` depends strongly on the time step. Both `rate` and `rate2` are initialized to -1, and stay that way until they are replaced, if they are replaced. All `rate2` values should be replaced and checked by `setrates` before a simulation is run. `rpar` is the reversible parameter for the potential reactivity of the products, and can take on any of several meanings, where its type is stored in `rpart`. See the discussion above for a list of the reversible types and parameters, as well as how they are used.

Following are some code fragments for traversing a reaction structure of arbitrary order. The former fragment walks through the reactions of all reactants and identifies the reaction number for each; the latter one walks through the reactions and identifies the molecule templates for each. To simplify them, the variables

order, maxident, nrxn, table, total, rate, nprod, and prod have been defined to be equal to the respective elements of a reaction structure or simulation structure.

```
ni2o=intpower(maxident,order);
for(i=0;i<ni2o;i++)
    for(j=0;j<nrxn[i];j++)
        r=table[i][j];

for(r=0;r<total;r++)
    for(p=0;p<nprod[r];p++)
        mptr=prod[r][p];
```

`rxnptr rxnalloc(int order,int maxident,int total);`
`rxnalloc` allocates and initializes a reaction structure, leaving it fully set up but with zero reactions. It can be used as is, but it won't do anything until reactions are added. `order` and `total` need to be at least 0 and `maxident` needs to be at least 1.

`void rxnfree(rxnptr rxn,int maxident);`
`rxnfree` frees a reaction structure for any order reaction.

`int loadrxn(simptr sim,FILE *fptr,int *lctrptr,char *erstr);`
`loadrxn` loads a reaction structure from an already opened disk file pointed to with `fptr`. `lctrptr` is a pointer to the line counter, which is updated each time a line is read. If successful, it returns 0 and the reaction is added to `sim`. Otherwise it returns the updated line counter along with an error message. If a reaction structure of the same order has already been set up, this function can use it and add more reactions to it. It can also allocate and set up a new structure, if needed. If this runs successfully, the complete reaction structure is set up, with the exception of `rate2` and the position vectors of the template molecules, which are all set to 0's (unless the product parameter type is 'o' or 'f', in which case they are set up). If the routine fails, the reaction structure is freed.

`void rxnoutput(simptr sim,int order);`
`rxnoutput` displays the complete contents of a reaction structure for order `order`. It also does some other calculations, such as the probability of geminate reactions for the products and the diffusion and activation limited rate constants.

The following routines work primarily with reaction structures and are intended to be run during program initialization, although after most of the reaction structures have been set up. They convert mass action reaction rates to microscopic simulation parameters and *vice versa*.

`int findreverserxn(simptr sim,int i1,int i2,int r,int *optr,int *rptr) {`
`findreverserxn` inputs the reaction defined by reactants `i1` and/or `i2` and reaction number `r` and looks to see if there is a reverse reaction. If both `i1` and `i2` are 0, then the forward reaction is zeroth order and there is no direct reverse reaction. If either `i1` or `i2` is 0, the forward reaction is first order, and if neither reactant is 0, the forward reaction is second order. If there is no reaction number `r`, an error code of

–1 is returned. If there is a direct reverse reaction, meaning the products of the input reaction are themselves able to react to form identities i1 and i2 (or just one of them if the input reaction is first order), then the function returns 1 and the order and reaction number of the reverse reaction are pointed to by optr and rptr. If there is no direct reverse reaction, but the products of the input reaction are still able to react, the function returns 2 and optr and rptr point to the first listed continuation reaction. If the products do not react, the function returns 0 (this also includes the situation where the products of the input reaction are A and B and there is no A+B reaction, although A and/or B can undergo unimolecular reactions, as well as all situations in which there are three or more products of a forward reaction). Either or both of optr and rptr are allowed to be sent in as NULL values if the respective pieces of output information are not of interest.

```
int setrates(simptr sim,int order);
```

setrates is used to convert the requested reaction rates to values that are useful for the simulation routines. Values in the rate element are read and values are written to rate2. If a rate element is less than zero, it is assumed to have been unassigned and is skipped; in this case, the respective value for rate2 is not modified. For zeroth order reactions, $rate_2$ is the expectation number of molecules that should be produced in the entire simulation volume during one time step, which is $rate*dt*volume$. For first order reactions, $rate_2$ is the probability of a unimolecular reaction occurring for an individual reactant molecule during one time step, which is $rate/sum*[1-\exp(-sum*dt)]$, where sum is the sum of the defined rate values for all unimolecular reactions of the reactant. For second order reactions, $rate_2$ is the squared binding radius of the reactants, found from bindingradius. In this case, the reverse parameter is accounted for in the reaction rate calculation if there is a direct reverse reaction and if it is appropriate (see the discussion of “Binding and unbinding radii,” above and the description for findreverserxn). This routine also sets the lists member of reaction structures. The return value of the function is –1 for correct operation. If errors occur, which is only possible for illegal inputs (return value of 0) or bimolecular calculations, the reaction number where the error was encountered is returned. Other than illegal inputs, the only possible errors arise from a diffusion constant of 0 for both reactants, or a directly reversible reaction that has an undeclared reversible type.

```
int setproducts(simptr sim,int order,char *erstr);
```

setproducts is used to set the initial separations between reaction products for all reactions of order order, based on the corresponding rpart character and rpar parameter. This is done by calculating the proper separation and then setting the first value of the template molecule pos vector to this separation, for all appropriate templates. If rpart is either ‘o’ or ‘f’, denoting either randomly oriented offset or fixed orientation offset, then it is assumed that the template molecule position has already been set up; it is not modified again by this routine. Otherwise, it is assumed that the template molecule position vectors have all values equal to 0 initially. If there are illegal inputs, 0 is returned by this routine. If an error occurs, the reaction number where the error was encountered is returned and a message is returned in the string erstr, which should have been allocated to size STRCHAR; if all

assignments work correctly, -1 is returned and the string is unchanged; and if a reversible reaction was undeclared, -1 is returned and a warning is returned in the string, although the program does not need to terminate. Possible errors include trying to set product positions for reactions without products, setting relative positions for products in which reactions have only one product, are irreversible, or have multiple reverse reactions, or trying to get a geminate binding probability that is unachievably high due to too long a time step. See the discussion in the section called "Binding and unbinding radii" for more details.

```
double calcrate(simptr sim,int i1,int i2,int r,double *pgemptr);
```

calcrate calculates the macroscopic rate constant using the microscopic parameters that have been calculated or that were initially assigned. All going well, these results should exactly match those that were requested initially, although this routine is useful as a check, and for situations where the microscopic values were input rather than the mass action rate constants. For bimolecular reactions that are reversible, the routine calculates rates with accounting for reversibility if the product parameter is p, x, r, b, or ?, and not otherwise. A value of -1 is returned if input parameters are illegal and a value of 0 is returned if the rate2 value for the indicated reaction is undefined (<0). If reversibility is accounted for and pgemptr is not input as NULL, *pgemptr is set to the probability of geminate recombination of the reactants; otherwise its value is not changed.

The following routines are used during simulation.

```
int doreact(rxnptr rxn,int r,moleculeptr mptr1,moleculeptr mptr2,simptr sim);
```

doreact executes a reaction that has already been determined to have happened. rxn is the reaction, r is the reaction number and mptr1 and mptr2 are the reactants, where mptr2 is ignored for unimolecular reactions, and both are ignored for zeroth order reactions. Reactants are killed, but left in the live lists. Any products are created on the dead list, for transfer to the live list by the molsort routine. Molecules that are created are put at the reaction position, which is the average position of the reactants weighted by the inverse of their diffusion constants, plus an offset from the product definition. The cluster of products is typically rotated to a random orientation. If the displacement was set to all 0's (recommended for non-reacting products), the routine is fairly fast, putting all products at the reaction position. If the rpart character is 'f', the orientation is fixed and there is no rotation. Otherwise, a non-zero displacement results in the choosing of random angles and vector rotations. If the system has more than three dimensions, only the first three are randomly oriented, while higher dimensions just add the displacement to the reaction position. The function returns 0 for successful operation and 1 if more molecules are required than were initially allocated. This function lists the correct box in the box element for each product molecule, but does not add the product molecules the molecule list of the box.

```
int zeroreact(simptr sim);
```

zeroreact figures out how many molecules to create for each zeroth order reaction and then tells doreact to create them. It returns the number of molecules created, or -1 if not enough molecules were allocated initially.

```
int unireact(simptr sim);
```

unireact determines whether unimolecular reactions occurred, considering both live lists. Reactions that do occur are sent to doreact to process them. The function returns the number of reactions that occurred during that time step, or -1 if not enough molecules were allocated initially.

```
int bireact(simptr sim,int neigh);
```

bireact determines whether bimolecular reaction occurred, sending ones that do occur to doreact. neigh tells the routine whether to consider only reactions between neighboring boxes (neigh=1) or only reactions within a box (neigh=0). The former are relatively slow and so can be ignored for qualitative simulations by choosing a lower simulation accuracy value. In cases where walls are periodic, it is possible to have reactions over the system walls. The function returns the number of reactions that occurred during that time step, or -1 if not enough molecules were allocated initially.

3.4 Surfaces

Surfaces are organized with a surface superstructure that contains not much more than just a list of surfaces and their names. Each of these surfaces, defined with a surface structure, has various properties that apply to the whole surface, such as its color on the front and back faces, how it is drawn, and how it interacts with diffusing molecules. A surface structure also includes lists of panels that comprise the surface. These panels may be rectangular, triangular, spherical, cylindrical, or hemispherical. A single surface can contain many panels of multiple shapes.

There is somewhat more to these panel shapes than their names might indicate. Rectangular panels are designed to be easy to use and to simulate efficiently, but are also limited. They are always perpendicular to an axis and have edges that are parallel to other axes. They are defined for all dimensions. Triangular panels are much more versatile but harder to use. The winding rule is that the front side has counter-clockwise winding, meaning that they obey the right-hand rule for winding. In 2-D, the front side is the right side of the line, when travelling in the sequence of listed points.

The table below lists the types of panels and key aspects of how they are stored internally. Each panel shape is described by either a character, such as 'r' for rectangle, or a number, such as 0 for rectangle. These descriptions, typically given with the variables pshape or ps, respectively, are in complete correspondence; translations from character to number can be done with the function pshape2ps. Panel locations and sizes, plus some drawing information, are given with sets of dim-dimensional points. There are npts points for a panel, listed below, where npts depends on both the panel shape and the system dimensionality; npts can also be gotten from pshape2ps. Additionally, each panel has a dim-dimensional front vector, which contains information about the direction that the panel faces. In some cases, such as for triangles, this is the normal vector to the

surface and is redundant with the information in the points. In others, it contains additional information. For example, for spheres, only one element of front is used, and it is used to tell if the front of the panel is on the inside or outside of the sphere, which cannot be known from just the list of points. In the table below, p is used for point, and f is used for front.

1D	2D	3D
rectangles ('r'), ps = 0		
npts = 1	npts = 2	npts = 4
p[0][0] = location	p[0][0...1] = start	p[0...3][0...2]
	p[1][0...1] = end	= corners
	parallel to an axis	parallel to an axis
	front is on right	front has CCW winding
f[0] = ±1	f[0] = ±1	f[0] = ±1
(+ for facing +0)	(+ for facing +axis)	(+ for facing +axis)
f[1] = 0 (perp. axis)	f[1] = perp. axis (0,1)	f[1] = perp. axis (0,1,2)
f[2] = undefined	f[2] = parallel axis	f[2] = axis parallel
		to edge from point 0 to point 1
<hr/>		
triangles ('t'), ps = 1		
npts = 1	npts = 2	npts = 3
p[0][0] = location	p[0][0...1] = start	p[0...2][0...2]
	p[1][0...1] = end	= corners
	front is on right	front has CCW winding
f[0] = ±1	f[0...1] = normal vect.	f[0...2] = normal vect.
(+1 for facing +0)		
<hr/>		
spheres ('s'), ps = 2		
npts = 2	npts = 2	npts = 2
p[0][0] = center	p[0][0...1] = center	p[0][0...2] = center
p[1][0] = radius	p[1][0] = radius	p[1][0] = radius
	p[1][1] = slices	p[1][1] = slices
		p[1][2] = stacks
f[0] = ±1	f[0] = ±1	f[0] = ±1
(+ for front outside)	(+ for front outside)	(+ for front outside)
f[1...2] = undefined	f[1...2] = undefined	f[1...2] = undefined
<hr/>		
cylinders ('c'), ps = 3		
	npts = 3	npts = 3
	p[0][0...1] = start center	p[0][0...2] = start center
not allowed	p[1][0...1] = stop center	p[1][0...2] = stop center
	p[2][0] = radius	p[2][0] = radius
		p[2][1] = slices
	f[0...1] = norm. right vect.	p[2][2] = stacks
	f[2] = ±1	f[0] = ±1
	(+ for front outside)	(+ for front outside)
		f[1...2] = undefined

hemispheres ('h'), ps = 4

	npts = 3	npts = 3
	p[0][0...1] = center	p[0][0...2] = center
not allowed	p[1][0] = radius	p[1][0] = radius
	p[1][1] = slices	p[1][1] = slices
		p[1][2] = stacks
	p[2][0...1] = outward vect.	p[2][0...2] = outward vect.
	f[0] = ± 1	f[0] = ± 1
	(+ for front outside)	(+ for front outside)
	f[1...2] = undefined	f[1...2] = undefined

```
typedef struct panelstruct {
    char pshape;          // panel shape: (r)ect, (t)riangle, (s)phere
    struct surfacestruct *srf; // surface that owns this panel
    int npts;             // number of defining points
    double **point;       // defining points, [number][dimension]
    double front[3];      // front parameters, which depend on the shape
} *panelptr;
```

pshape is character 'r', 't', 's', 'c', or 'h' for a rectangular, triangular, spherical, cylindrical, or hemispherical panel, respectively. As described in the user documentation, these names only pertain to 3D simulations, but most of them have shape analogs in 1D and 2D. srf is a pointer to the surface that owns this panel; it would be called a surfaceptr, except that a surfaceptr isn't declared until later. npts is the number of dim-dimensional points that are allocated. points and front have meanings that depend on the panel shape and on the dimensionality, described in the preceding table.

```
typedef struct surfacestruct {
    char *faction;        // action for molec. on front [maxident]
    char *baction;        // action for molecules on back [maxident]
    double fcolor[4];     // RGBA color vector for front
    double bcolor[4];     // RGBA color vector for back
    double edgepts;       // thickness of edge for drawing
    char fpolymode;       // polygon drawing mode for front
    char bpolymode;       // polygon drawing mode for back
    int maxpanel[PSMAX];  // allocated number of panels [ps]
    int npanel[PSMAX];    // actual number of panels [ps]
    panelptr *panels[PSMAX]; // list of panels [ps][p]
} *surfaceptr;
```

faction and baction are the actions that happens to molecules that diffuse into the front or back face of the surface, respectively, which can be 'r' for reflect, 'a' for absorb, or 't' for transparent. These vectors have maxident elements to account for each molecule species. fcolor and bcolor are the colors of the front and back of the surface in the order: red, green, blue, alpha; each has a value between 0 and 1. edgepts is the thickness of edges in points for drawing, which applies to all drawing

situations except for 3D and when the surface faces are rendered. `fpolymode` and `bpolymode` are characters that describe how the surface front and back should be drawn: 'v' for vertices only, 'e' for edges only, or 'f' for faces only. Not all options apply to 1D and 2D simulations. `maxpanel` and `npanel` are the number of panels that are allocated or used, respectively, for each of the panel shapes. `panels` are lists of pointers to the panels for the possible shapes. Note that every panel within a surface has the same drawing scheme and the same interaction with molecules.

```
typedef struct surfacesuperstruct {
    int maxsrf;           // maximum number of surfaces
    int nsrf;             // number of surfaces
    char **snames;        // surface names [s]
    surfaceptr *srflist;  // list of surfaces [s]
    int drctonly;         // 1 for only detect direct collisions
} *surfacessptr;
```

`maxsrf` and `nsrf` are the number of surfaces that are allocated and defined, respectively. `snames` is a list of names for the surfaces. `srflist` is the list of pointers to surfaces.

It was surprisingly difficult to get the surfaces to work well enough that diffusing molecules did not leak through reflective panels on rare occasion. Because of that, the code is written unusually carefully, and in ways that are not necessarily obvious, so be careful when modifying it. For example, round-off error differences between two different but mathematically identical ways of calculating a molecule distance from a surface can easily place the molecule on the wrong side of a surface panel.

If a molecule is exactly at a panel, it is considered to be at the back side of the panel. Initially, I defined direct collisions as collisions in which the straight line between two points crosses a surface, whereas an indirect collision is one in which the straight line does not cross a surface but it was determined with a random number that the Brownian motion trajectory did contact the surface. Indirect collisions proved to slow down the program significantly, greatly complicate the code development, and provided minimal accuracy improvements, so I got rid of them. Now, only direct collisions are detected and dealt with.

Currently, surface actions are reflect ('r'), absorb ('a'), transparent ('t'), or periodic ('p'). In the future, it will also be possible to combine the first three in various ways such that a surface can be semi-permeable and/or weakly adsorbing. Periodic surfaces are designed solely to provide periodic outside boundaries of the system.

```
int pshape2ps(char pshape,int dim,int *nptsptr);
```

Converts panel shape character in `pshape` to the panel shape number, which is returned. Also, if `nptsptr` is not sent in a NULL, the number of points that are used for this panel shape and for the dimensionality that is entered in `dim` is returned in it. If `pshape` is not a recognized character or if the panel shape is not supported for this dimensionality, -1 is returned and `nptsptr` is set to point to a 0.

`int panelsalloc(surfaceptr srf,int dim,int maxpanel,char pshape);`
 Allocates maxpanels of shape pshape for the surface srf. The srf element of the panels are set to srf. In srf, the correct maxpanel entry is set to maxpanel, the npanel entry is set to 0, and the proper list of panels are allocated and cleared (srf can also be sent in as NULL, in which case, these are clearly not done). All points are set to all zeros. The function returns 1 for success and 0 for failure to allocate memory.

`void panelfree(panelptr pnl);`
 Frees a single panel and all of its substructures (but not srf, because that's a reference and is not owned by the panel). This is called by surfacefree and so should not need to be called externally.

`surfaceptr surfacealloc(int maxident);`
 Allocates a surface structure, and sets all elements to initial values. maxident is the maximum number of molecular identities, which is used for allocating faction and baction. Colors are set to all 0's (black), but with alpha values of 1 (opaque); polygon modes are set to 'f' for face; edgepoints is set to 1; faction and baction are set to 'r' for reflective. No panels are allocated. This is called by surfacessalloc and so should not need to be called externally.

`void surfacefree(surfaceptr srf);`
 Frees a surface, including all substructures and panels in it. This is called by surfacessfree and so should not need to be called externally.

`surfacessptr surfacessalloc(int maxsurface,int maxident);`
 Allocates a surface superstructure for maxsurface surfaces, as well as all of the surfaces. Each name is allocated to an empty string of STRCHAR (256) characters.

`void surfacessfree(surfacessptr srfss);`
 Frees a surface superstructure pointed to by srfss, and all contents in it, including all of the surfaces and all of their panels.

`int loadsurface(simptr sim,FILE *fptr,int *lctrptr,char *erstr);`
 loadsurface loads a surface from an already opened disk file pointed to with fptr. lctrptr is a pointer to the line counter, which is updated each time a line is read. If successful, it returns 0 and the surface is added to the surface superstructure in sim, which should have been already allocated. Otherwise it returns the updated line counter along with an error message. If a surface with the same name (entered by the user) already exists, this function can add more panels to it. It can also allocate and set up a new surface. If this runs successfully, the complete surface structure is set up, with the exception of box issues. If the routine fails, any new surface structure is freed.

`void surfaceoutput(simptr sim);`
 Prints out information about all surfaces, including the surface superstructure, each surface, and panels in the surface.

`char panelside(double* pt,panelptr pnl,int dim);`

Returns the side of the panel `pnl` that point `pt` is on, which is either a 'f' or a 'b' for front or back, respectively. 'b' is returned if the point is exactly at the panel. The value returned by this function defines the side that `pt` is on, so should either be called or exactly copied for other functions that care.

`int lineXpanel(double *pt1,double *pt2,panelptr pnl,char *faceptr,double *crsspt,double *cross,int dim);`

This determines if the line from `pt1` to `pt2` crosses the panel `pnl`, using a `dim` dimensional system. The panel includes all of its edges. If it crosses, 1 is returned, the face that is on the `pt1` side of the panel is returned in `faceptr`, `crsspt` is set to the coordinates of the crossing point, and `cross` points to the crossing position on the line, which is a number between 0 and 1, inclusive. If it does not cross, 0 is returned and the other values are undefined. Crossing is handled very carefully such that the exact locations of `pt1` and `pt2`, using tests that are identical to those in `panelside`, are used to determine which sides of the panel they are on. While `crsspt` will be returned with coordinates that are very close to the panel location, it may not be precisely at the panel, and there is no certainty about which side of the panel it will be on; if it matters, fix it with `fixpt2panel`.

If the line crosses the panel more than once, which can happen for spherical or other curved panels, the smaller of the two crossing points is returned. For sphere and cylinder, 0 is returned if either both points are inside or both points are outside and the line segment does not cross the object twice.

Each portion of this routine does the same things, and usually in the same order. First, the potential intersection face is determined, then the crossing value, then the crossing point, and finally it finds if intersection actually happened. For hemispheres and cylinders, if intersection does not happen for the first of two possible crossing points, it is then checked for the second point.

`int rxnXsurface(simptr sim,moleculeptr mptr1,moleculeptr mptr2);`

Returns 1 if a potential bimolecular reaction between `mptr1` and `mptr2` is across a non-transparent surface, and so cannot actually happen. Returns 0 if a reaction is allowed. Using the diffusion coefficients of the two molecules, this calculates the reaction location and then determines which molecule needs to diffuse across a surface to get to that location. If that molecule can diffuse across the surface, then the reaction is allowed, and not otherwise.

`void fixpt2panel(double *pt,panelptr pnl,int dim,char face);`

Fixes the point `pt` onto the face `face` of panel `pnl`. Send in `face` equal to '0' if `pt` should be moved as close as possible to `pnl`. If it should also be on the front or back face of the panel, as determined by `panelside`, then send in `face` equal to 'f' or 'b', respectively. This function first moves `pt` to the panel in a direction normal to the local panel surface and then nudges `pt` as required to get it to the proper side. This only considers the infinite plane of the panel, while ignoring its boundaries

(similarly, hemispheres are considered to be identical to spheres and cylinders are considered to be infinitely long).

```
void surfacereflect(moleculeptr mptr,panelptr pnl,double *crsspt,char face,int dim);
```

This bounces the molecule `mptr` off of the `face` side of panel `pnl`. Elastic collisions are performed, which should work properly for any shape panel and any dimensionality. For flat panels, elastic collisions also apply to Brownian motion. It is assumed that the molecule travels from some point (not given to this function, and irrelevant) to `mptr->pos`, via a collision with the panel at location `crsspt`, where `crsspt` is either exactly at the panel or is slightly on impact side of the panel. The molecule `pos` element is set to the new, reflected, position, which will always be on the `face` side of the panel.

```
int dosurfinteract(moleculeptr mptr,panelptr pnl,char face,double *crsspt,int dim);
```

Performs interaction between molecule and surface for an interaction that is known to have happened. If the surface is reflective and it is found that `crsspt` is not on the same side of the panel as `via`, then `crsspt` is moved slightly so that it is on the same side as `via`. This calls `surfacereflect` if needed. It also absorbs a molecule if needed. Returns 1 if the molecule does not need additional trajectory tracking (e.g. it's absorbed) and 0 if it might need additional tracking (e.g. it's reflected).

```
int checksurfaces(simptr sim,int ll,int reborn);
```

Takes care of interactions between molecules and surfaces. Molecules in live list `ll` are considered; if `reborn` is 1, only the reborn molecules of list `ll` are considered. This transmits, reflects, or absorbs molecules, as needed, based on the panel positions and information in the molecule `posx` and `pos` elements. Absorbed molecules are killed but left in the live list with an identity of zero, for later sorting. Reflected molecules are bounced and their `posx` values represent the location of their last bouncing point. This function does not rely on molecules being properly assigned to boxes, and nor does it assign molecules to boxes afterwards. However, it does rely on the panels being properly assigned to boxes. The total number of surface interactions is returned.

3.5 Boxes

The simulation volume is exactly divided into an array of identical virtual boxes. These allow the simulation to run efficiently because only potential reactions between molecules that are known to be physically close need to be checked, and the same for molecule-surface interactions. In principle, the boxes are fairly simple. In practice though, they complicate the overall code quite significantly.

Each box has its own box structure. These are collected in a box superstructure.

```
typedef struct boxstruct {  
    int *indx;                                dim dimensional index of the box
```

<code>int nneigh;</code>	number of neighbors in list
<code>int midneigh;</code>	logical middle of neighbor list
<code>struct boxstruct **neigh;</code>	all box neighbors, using sim. accuracy
<code>int *wpneigh;</code>	wrapping code of neighbors in list
<code>int nwall;</code>	number of walls in box
<code>wallptr *wlist;</code>	list of walls that cross the box
<code>int npanel;</code>	number of surface panels in box
<code>panelptr *panel;</code>	list of panels in box
<code>int maxmol[2];</code>	allocated size of live lists
<code>int nmol[2];</code>	number of molecules in live lists
<code>moleculeptr *mol[2]; } *boxptr;</code>	lists of live molecules in the box

boxstruct (declared in `smollib.h`) is a structure for each of the virtual boxes that partition space. Each box has a list of its neighbors, in `neigh`, as well as a little information about them. This list extends from 0 to `nneigh-1`. From 0 to `midneigh-1` are those neighbors that logically precede the box, meaning that they are above or to the left, whereas those from `midneigh` to `nneigh-1` logically follow the box. If there are no periodic boundary conditions, the logical order is the same as the address order; however, this is not necessarily true with the inclusion of wrap-around effects. In `wpneigh` is a code for each neighbor that describes in what way it is a neighbor: 0 means that it's a normal neighbor with no edge wrap-around; otherwise pairs of bits are associated with each dimension (low order bits for low dimension), with the bits equal to 00 for no wrapping in that dimension, 01 for wrapping towards the low side, and 10 for wrapping towards the high side. This might be clearer in the Zn.c documentation. The neighbors that are listed depend on the requested simulation accuracy:

<u>accuracy</u>	<u>neighbors</u>	<u>wrap-around</u>
<3	none	no
3 to <6	nearest	no
6 to <9	nearest	yes
>9	all	yes

Boxes also have lists of mobile molecules (`mol[0]`, allocated to size `maxmol[0]`, and filled from 0 to `nmol[0]-1`), immobile molecules (`mol[1]`, etc.) molecules, and walls (`wlist`, allocated and filled with `nwall` pointers) within them. While the lists are owned by the box, the members of the lists are simply references, rather than implications of ownership. The same, of course, is true of the neighbor list, although the box owns the `wpneigh` list. If wall or neighbor lists are empty, the list is left as `NULL`, whereas the molecule list always has a few spaces in it.

<code>typedef struct boxsuperstruct {</code>	
<code>double mpbox;</code>	requested number of molecules per box
<code>double boxsize;</code>	requested box width
<code>int nbox;</code>	total number of boxes
<code>int *side;</code>	number of boxes on each side of space

<code>double *min;</code>	position vector for low corner of space
<code>double *size;</code>	length of each side of a box
<code>boxptr *blist; } *boxssptr;</code>	actual array of boxes

`boxsuperstruct` (declared in `smollib.h`) expresses the arrangement of virtual boxes in space, and owns the list of those boxes and the boxes. Either `mpbox` or `boxsize` are used and not both. Boxes are arranged in a rectanguloid grid and exactly cover all space inside the walls. The structure of the boxes in space is the same as that of a `dim` rank tensor, allowing tensor indexing routines to be used to convert between box addresses and indicies. The box index along the d 'th dimension of a point with position `x[d]` is

```
indx[d]=(x[d]-min[d])/size[d];
```

where integer arithmetic takes care of the truncation. Converging from box index to address is easy with the tensor routine in `Zn.c`, or can also be calculated quickly with the following code fragment, which outputs the box number as `b`,

```
for(b=0,d=0;d<dim;d++) b=side[d]*b+indx[d];
```

Converting the box number to the indicies can also be done, but the `Zn.c` routine is easiest for this.

```
boxptr boxalloc(int dim);
```

`boxalloc` allocates and minimally initializes a new `boxstruct`. The only list allocated is `indx`, which is set to 0's.

```
void boxfree(boxptr bptr);
```

`boxfree` frees the box and its lists, although not the structures pointed to by the lists.

```
boxptr *boxesalloc(int dim,int nbox);
```

`boxesalloc` allocates and initializes an array of `n` boxes, including the boxes. Again, initialization is minimal, with only the `indx` array of the boxes allocated, which is set to 0's.

```
void boxesfree(boxptr *blist,int nbox);
```

`boxesfree` frees an array of boxes, including the boxes.

```
boxssptr boxssalloc(int dim);
```

`boxssalloc` allocates and initializes a superstructure of boxes, including arrays for the `side`, `min`, and `size` members, although the boxes are not added to the structure; *i.e.* `blist` is set to `NULL` and `nbox` is 0. Initial values for `side` and `size` members are all set to 1, `min` values are set to 0, and `mpbox` is set to 5; all of these values are typically changed later.

```
void boxssfree(boxssptr boxss);
```

`boxssfree` frees a box superstructure, including the boxes.

`void boxoutput(int dim,boxssptr boxs);`
 boxoutput simply lists every virtual box, along with all the details about it, where these details are the index, the number of neighbors, the neighbor mid-point, the number of maximum number of molecules, what the neighbors are, and what the wrapping codes are. As the program is currently written, this function is never called, although it could be to look for errors in box setting up.

`void boxssoutput(simptr sim);`
 boxssoutput displays statistics about the box superstructure, including total number of boxes, number on each side, dimensions, and the minimum position. It also prints out the requested and actual numbers of molecules per box.

`int expandbox(boxptr bptr,int n,int ll);`
 expandbox is called if it turns out that a box was not allocated with enough space for molecules. bptr is a pointer to a box that needs expanding, n is the number of additional molecule spaces to add, and ll is the live list to expand. If n is negative, the box is shrunk and any molecule pointers that no longer fit are simply left out. The book keeping elements of the box are updated. The function returns 0 if it was successful and 1 if there was not enough memory for the request.

`boxptr pos2box(simptr sim,double *pos);`
 pos2box returns a pointer to the box that includes the position given in pos, which is a dim size vector. If the position is outside the simulation volume, a pointer to the nearest box is returned. This routine assumes that the entire box superstructure is set up.

`void box2pos(simptr sim,boxptr bptr,double *pos);`
 Given a pointer to a box in bptr, this returns the coordinate of the low corner of the box in pos, which needs to be pre-allocated to the system dimensionality. This requires that the min and size portions of the box superstructure have been already set up. To get the other box corners, add the size values that are in the box superstructure.

`boxptr line2nextbox(simptr sim,double *pt1,double *pt2,boxptr bptr);`
 Given a line segment which is defined by the starting point pt1 and the ending point pt2, and which is known to intersect the virtual box pointed to by bptr, this returns a pointer to the next box along the line segment. If the current box is also the final one, NULL is returned. NULL is also returned if the next box is outside of the region that is covered by virtual boxes.

`int panelinbox(simptr sim,panelptr pnl,boxptr bptr);`
 Determines if any or all of the panel pnl is in the box bptr and returns 1 if so and 0 if not. For most panel shapes, this is sufficiently complicated that this function just calls other functions in the library file Geometry.c.

`int setupboxes(simptr sim);`

setupboxes sets up a superstructure of boxes, and puts things in the boxes, including wall and molecule references. It requires a simulation structure with most things set up, but not box stuff; it's designed for the structure after it's returned from loadsimul. It sets up the box superstructure, then adds indices to each box, then adds the box neighbor list along with neighbor parameters, then adds wall references to each box, and finally creates molecule lists for each box and sets both the box and molecule references to point to each other. The molecule list is $6+3\sqrt{n_{mol}}$ higher than n_{mol} to allow for more molecules; the extra pointers are all set to NULL. The function returns 0 for successful operation and 1 if it was unable to allocate sufficient memory. At the end, all simulation parameters having to do with boxes are set up. However, some lists may still be NULL, if they are empty, where these are `bptr->neigh`, `bptr->wpneigh`, and `bptr->wlist`. Of particular note is that `bptr->wpneigh` is NULL if no neighbors are wrap-around ones, for whatever reason.

```
int assignmolecs(simptr sim,int ll);
```

`assignmolecs` puts molecules in boxes by overwriting the lists of molecules that are in each box with molecules from `mols`. It only assigns the live list number `ll`. Molecules that are outside the set of boxes are assigned to the nearest box. If more molecules belong in a box than actually fit, 5 more spaces are allocated using `expandbox`. The function returns 0 unless memory could not be allocated by `expandbox`, in which case only some of the molecules are assigned and it returns 1.

```
int reassignmolecs(simptr sim,int ll,int reborn);
```

Updates the information about which molecule is in which box, for those molecules that are in master list `ll` (0 for diffusing, 1 for not). If `reborn` is 0, all molecules in list `ll` are updated; otherwise, only the reborn ones are. This assumes that all molecules that are being checked were in a box, meaning that the box element of the molecule structure listed a box and that the `mol` list of that box listed the molecule. Molecules are arranged in boxes according to the location of the `pos` element of the molecules. Molecules outside the set of boxes are assigned to the nearest box. If more molecules belong in a box than actually fit, the number of spaces is doubled using `expandbox`. The function returns 0 unless memory could not be allocated by `expandbox`, in which case it fails and returns 1.

3.6 Simulation structure

At the highest level of the structures is the simulation structure. This is a large framework that contains information about the simulation that is to be run as well as pointers to each of the component structures and superstructures. It also contains some scratch space for functions to use as they wish.

```
typedef struct simstruct {
    unsigned int randseed;           random number generator seed.
    int dim;                         dimensionality of space.
    int maxident;                    maximum number of identities
    int nident;                      number of identities, including empty mols
}
```

<code>char **name;</code>	names of molecules
<code>int graphics;</code>	graphics: 0=none, 1=opengl, 2=good opengl
<code>int graphicit;</code>	number of time steps per graphics update
<code>int tiffit;</code>	number of time steps per tiff save
<code>double framepts;</code>	thickness of frame for graphics
<code>double gridpts;</code>	thickness of virtual box grid for graphics
<code>double framecolor[4];</code>	frame color
<code>double backcolor[4];</code>	background color
<code>double accur;</code>	accuracy, on scale from 0 to 10
<code>double time;</code>	current time in simulation
<code>double tmin;</code>	simulation start time
<code>double tmax;</code>	simulation end time
<code>double dt;</code>	simulation time step
<code>rxnptr rxn[3];</code>	list of reactions
<code>molssptr mols;</code>	molecule superstructure
<code>wallptr *wlist;</code>	list of walls
<code>surfacessptr srfss;</code>	surface superstructure
<code>boxssptr boxes;</code>	box superstructure
<code>cmdssptr cmds;</code>	command superstructure
<code>double *v1,*v2,*v3;</code>	scratch space, each size dim or maxident
<code>double *m1,*m2,*m3;</code>	scratch space, each size dim x dim
<code>int *z1,*z2,*z3; } *simptr;</code>	scratch space, each size dim or maxident

`simstruct` (declared in `smollib.h`) contains and owns all information that defines the simulation conditions, the current state of the simulation, and all other simulation parameters. The scratch space is allocated when the structure is allocated and is for the use of any routine that uses a simulation structure. The `v` and `z` scratch space vectors have dimensions that are the larger of `dim` or `maxident`.

`simptr simalloc(int dim,int maxident,char *root);`
`simalloc` allocates a simulation structure. The `difc` and `difm` lists are allocated and initialized. Default diffusion matrices are all 0's, except with -1 in the first element. Walls are allocated and inialized. The box superstructure is allocated and initialized, although the list of boxes is left as `NULL`. The molecule superstructure is left as `NULL`. The commands superstructure is allocated, but the queue of commands and the output file lists are left as `NULL`s. `root` is required for the command superstructure. `simalloc` also sets the random number generator seed.

`void simfree(simptr sim);`
`simfree` frees a simulation strucutre, including every part of everything in it.

Initialization procedures are meant to be called once at the beginning of the program to allocate and set up the necessary structures. These routines call memory allocation procedures as needed. `setupstructs` is the only one of these routines that should ever need to be called externally, since it calls the other functions as needed.

`int loadsimul(simptr *smptr,char *fileroot,char *filename,char *erstr);`

loadsimul loads all simulation parameters from a configuration file, using a format described above. `fileroot` is sent in as the root of the filename, including all colons, slashes, or backslashes; if the configuration file is in the same directory as *Smoldyn*, `fileroot` should be an empty string. `filename` is sent in as just the file name and any extension. `erstr` is sent in as an empty string of size `STRCHAR` and is returned with an error message if an error occurs. `smptr` is sent in as a pointer to the variable that will point to the `simstruct`; it is returned pointing to a pointer to an initialized `simstruct`. This routine calls `loadrxn` to load in any reactions and `loadsurface` to load any surfaces. The following things are set up after this routine is completed: all molecule elements except `box`; all molecule superstructure elements; all wall elements; `box` superstructure element `mpbox`, but no other elements; no boxes are allocated or set up; all reaction structure elements except `rate2` and the product template position vectors (`pos` in each product); the command superstructure, including all of its elements; and all simulation structure elements except for sub-elements that have already been listed. All new molecules are left in the dead list for sorting later. If the configuration file loads successfully, the routine returns 0. If the file could not be found, it returns 10 and an error message. If an error was caught during file loading, the return value is 10 plus the line number of the file with an error, along with an error message. If there is an error, all structures are freed automatically.

```
int setupstructs(char *root, char *name, simptr *smptr, int vb);
```

`setupstructs` sets up and loads values for all the structures as well as global variables. This routine calls the other initialization routines, so they do not have to be called from elsewhere. Other minor things are set up here, including setting the lookup table for normally distributed random numbers. It also displays the status to `stdout` and calls output routines for each structure, allowing verification of the initialization. Send in `root` and `name` with strings for the path and name of the input file. `vb` is a flag for verbose operation, 1 for verbose and 0 for quiet. It returns 0 for correct operation and 1 for an error. If it succeeds, `smptr` is returned pointing to a fully set up simulation structure. Otherwise, `smptr` is set to `NULL` and an error messages is displayed on `stderr`.

```
void simoutput(simptr sim);
```

`simoutput` prints out the overall simulation parameters, including simulation time information, graphics information, the number of dimensions, what the molecule types are, the output files, and the accuracy.

```
void checkparams(simptr sim);
```

`checkparams` checks that the simulation parameters, including parameters of sub-structures, have reasonable values. If values seem to be too small or too large, a warning is displayed to the standard output, although this does not affect continuation of the program.

4. Functions in `smolib2.c` (command interpreter routines)

4.1 Externally accessible function

Command strings are not parsed, checked, or even looked at during simulation initialization. Instead, they are run by the command interpreter during the simulation. Command routines are given complete freedom to look at and/or modify any part of a simulation structure or sub-structure. This, of course, also gives commands the ability to crash the computer program, so they need to be written carefully to prevent this. Every command is sent a pointer to the simulation structure in `sim`, as well as a string of command parameters in `line2`. See below for how to add a new command.

```
int docommand(void *cmdfnarg,cmdptr cmd,char *line);
```

`docommand` is given the simulation structure in `sim`, the command to be executed in `cmd`, and a line of text which includes the entire command string. It parses the line of text only into the first word, which specifies which command is to be run, and into the rest of the line, which contains the command parameters. The rest of the line is then sent to the appropriate command routine as `line2`. The return value of the command that was called is passed back to the main program from `docommand`. These routines return 0 for normal operation, 1 for an error that does not require simulation termination, 2 for an error that requires simulation termination, and 3 for a time step termination but no simulation termination (for pausing).

4.2 Individual command functions

```
int cmdstop(simptr sim,cmdptr cmd,char *line2);
int cmdpause(simptr sim,cmdptr cmd,char *line2);
int cmdoverwrite(simptr sim,cmdptr cmd,char *line2);
int cmdincrementfile(simptr sim,cmdptr cmd,char *line2);
int cmdifno(simptr sim,cmdptr cmd,char *line2);
int cmdifless(simptr sim,cmdptr cmd,char *line2);
int cmdifmore(simptr sim,cmdptr cmd,char *line2);
int cmdpointsources(simptr sim,cmdptr cmd,char *line2);
int cmdkillmol(simptr sim,cmdptr cmd,char *line2);
int cmdkillmolinsphere(simptr sim,cmdptr cmd,char *line2);
int cmdequilmol(simptr sim,cmdptr cmd,char *line2);
int cmdreplacexyzmol(simptr sim,cmdptr cmd,char *line2);
int cmdmodulatamol(simptr sim,cmdptr cmd,char *line2);
int cmdreact1(simptr sim,cmdptr cmd,char *line2);
int cmdmolcount(simptr sim,cmdptr cmd,char *line2);
int cmdlistmols(simptr sim,cmdptr cmd,char *line2);
int cmdlistmols2(simptr sim,cmdptr cmd,char *line2);
int cmdlistmols3(simptr sim,cmdptr cmd,char *line2);
int cmdmolpos(simptr sim,cmdptr cmd,char *line2);
int cmdmolmoments(simptr sim,cmdptr cmd,char *line2);
int cmdsavesim(simptr sim,cmdptr cmd,char *line2);
int cmdexcludebox(simptr sim,cmdptr cmd,char *line2);
int cmdexcludesphere(simptr sim,cmdptr cmd,char *line2);
int cmdincludeecoli(simptr sim,cmdptr cmd,char *line2);
```

```

int insideecoli(double *pos,double *ofst,double rad,double length);
void putinecoli(double *pos,double *ofst,double rad,double length);

int cmdstop(simptr sim,cmdptr cmd,char *line2);
    cmdstop returns a value of 2, meaning that the simulation should stop. Any contents
    of line2 are ignored.

int cmdpause(simptr sim,cmdptr cmd,char *line2);
    cmdpause causes the simulation to pause until the user tells it to continue.
    Continuation is effected by either pressing the space bar, if OpenGL is used for
    graphics, or by pressing enter if output is text only. The return value is 0 for non-
    graphics and 3 for graphics. Any contents of line2 are ignored.

int cmdoverwrite(simptr sim,cmdptr cmd,char *line2);
    cmdoverwrite overwrites a prior output file. See the user manual.

int cmdincrementfile(simptr sim,cmdptr cmd,char *line2);
    cmdincrementfile closes a file, increments the name and opens that one for output.
    See the user manual.

int cmdifno(simptr sim,cmdptr cmd,char *line2);
    cmdifno reads the first word of line2 for a molecule name and then checks the
    appropriate simulation live list to see if any molecules of that type exist. If so, it
    does nothing, but returns 0. If not, it sends the remainder of line2 to docommand to
    be run as a new command, and then returns 0. It returns 1 if the molecule name was
    missing or not recognized.

int cmdifless(simptr sim,cmdptr cmd,char *line2);
    cmdifless is identical to cmdifno, except that it runs the command in line2 if there
    are less than a listed number of a kind of molecules in the appropriate live list.

int cmdifmore(simptr sim,cmdptr cmd,char *line2);
    cmdifmore is identical to cmdifno except that it runs the command in line2 if there
    are more than a listed number of a kind of molecules in the appropriate live list.

int cmdpointsource(simptr sim,cmdptr cmd,char *line2);
    cmdpointsource reads line2 for a molecule name, followed by the number of
    molecules that should be created, followed by the dim dimensional position for
    them. If all reads well, it creates the new molecules in the system at the appropriate
    position. They are added to the dead list and then the lists are sorted.

int cmdkillmol(simptr sim,cmdptr cmd,char *line2);
    cmdkillmol reads line2 for a molecule name and then kills all molecules of that
    name from the appropriate live list by setting their identities to 0. The molecule
    lists are then sorted.

int cmdkillmolinsphere(simptr sim,cmdptr cmd,char *line2);

```

cmdkillmolinsphere reads line2 for a molecule name and a surface name and then kills all molecules of the given type, that are in spheres of the listed surface, from the appropriate live list by setting their identities to 0. The molecule lists are then sorted. The molecule name and/or the surface name can be “all”.

int cmdequilmol(simptr sim,cmdptr cmd,char *line2);
cmdequilmol equilibrates a pair of molecular species, allowing the efficient simulation of rapid reactions. It reads two molecule names from line2, followed by a probability value. Then, it looks for all molecules in the live lists with either of the two types and replaces them with the second type using the listed probability or with the first type using 1– the listed probability.

int cmdreplacexyzmol(simptr sim,cmdptr cmd,char *line2);
cmdreplacexyzmol reads the name of a molecule following by a dim dimensional point in space from line2. Then, it searches the fixed live list for any molecule that is exactly at the designated point. If it encounters one, it is replaced by the listed molecule, and then the live lists are sorted if appropriate. This routine stops searching after one molecule has been found, and so will miss additional molecules that are at the same point.

int cmdmodulatemol(simptr sim,cmdptr cmd,char *line2);
cmdmodulatemol is identical to cmdequilmol except that the equilibration probability is not fixed, but is a sinusoidally varying function. After reading two molecule names from line2, this routine then reads the cosine wave frequency and phase shift, then calculates the probability using the function $\text{prob}=0.5*(1.0-\cos(\text{freq}*\text{sim->time}+\text{shift}))$.

int cmdreact1(simptr sim,cmdptr cmd,char *line2);
cmdreact1 reads line2 for the name of a molecule followed by the name of a unimolecular reaction. Then, every one of that type of molecule is caused to undergo the listed reaction, thus replacing each one by reaction products. Molecules are sorted at the end. This might be useful for simulating a pulse of actinic light, for example.

int cmdsetrateint(simptr sim,cmdptr cmd,char *line2);
This reads line2 for the name of a reaction and the new internal rate constant for it. The internal rate constant is set to the new value. Errors can arise from illegal inputs, such as the reaction not being found or a negative requested internal rate constant.

int cmdmolcount(simptr sim,cmdptr cmd,char *line2);
cmdmolcount reads the output file name from line2. Then, to this file, it saves one line of text listing the current simulation time, followed by the number of each type of molecule in the system. This routine does not affect any simulation parameters.

int cmdlistmols(simptr sim,cmdptr cmd,char *line2);

cmdlistmols reads the output file name from line2. To this file, it saves a list of every individual molecule in both live lists of the simulation, along with their positions. This routine does not affect any simulation parameters.

int cmdlistmols2(simptr sim,cmdptr cmd,char *line2);
cmdlistmols2 reads the output file name from line2. To this file, it saves the number of times this command was invoked using the invoke element of commands, a list of every individual molecule in both live lists of the simulation, along with their positions. This routine does not affect any simulation parameters. Routine originally written by Karen Lipkow and then rewritten by me.

int cmdlistmols3(simptr sim,cmdptr cmd,char *line2);
cmdlistmols3 reads a molecule name and the output file name from line2. To this file, it saves the number of times the command was invoked, the identity of the molecule specified, and the positions of every molecule of the specified type. This routine does not affect any simulation parameters.

int cmdmolpos(simptr sim,cmdptr cmd,char *line2);
cmdmolpos reads a molecule name and then the output file name from line2. To this file, it saves one line of text with the positions of each molecule of the listed identity. This routine does not affect any simulation parameters.

int cmdmolmoments(simptr sim,cmdptr cmd,char *line2);
cmdmolmoments reads a molecule name and then the output file name from line2. To this file, it saves in one line of text: the time and the zeroth, first, and second moments of the distribution of positions for all molecules of the type listed. The zeroth moment is just the number of molecules (of the proper identity); the first moment is a *dim* dimensional vector for the mean position; and the second moment is a *dimxdim* matrix of variances. This routine does not affect any simulation parameters.

int cmdsavesim(simptr sim,cmdptr cmd,char *line2);
cmdsavesim reads the output file name from line2 and then saves the complete state of the system to this file, as a configuration file. This output can be run later on to continue the simulation from the point where it was saved.

int cmdexcludebox(simptr sim,cmdptr cmd,char *line2);
cmdexcludebox allows a region of the simulation volume to be effectively closed off to molecules. The box is defined by its low and high corners, which are read from line2. Any molecule, of any type, that entered the box during the last time step, as determined by its pos and posx structure members, is moved back to its previous position. This is not the correct behavior for a reflective surface, but is efficient and expected to be reasonably accurate for most situations. This routine ought to be replaced with a proper treatment of surfaces in the main program (rather than with interpreter commands), but that's a lot more difficult.

int cmdexcludesphere(simptr sim,cmdptr cmd,char *line2);

`cmdexcludesphere` is like `cmdexcludebox` except that it excludes a sphere rather than a box. The sphere is defined by its center and radius, which are read from `line2`. Any molecule, of any type, that entered the sphere during the last time step, as determined by its `pos` and `posx` structure members, is moved back to its previous position. This is not the correct behavior for a reflective surface, but is efficient and expected to be reasonably accurate for most situations.

```
int cmdincludeecoli(simptr sim,cmdptr cmd,char *line2);
```

`cmdincludeecoli` is the opposite of the `excludebox` and `excludesphere` commands. Here, molecules are confined to an *E. coli* shape and are put back inside it if they leave. See the user manual for more about it. Unlike the other rejection method commands, this one works even if a molecule was in a forbidden region during the previous time step; in this case, the molecule is moved to the point on the *E. coli* surface that is closest. Because of this difference, this command works reasonably well even if it is not called at every time step.

```
void cmdmeansqrdispfree(cmdptr cmd);
```

A memory freeing routine for memory that is allocated by `cmdmeansqrdisp`.

```
int cmdmeansqrdisp(simptr sim,cmdptr cmd,char *line2);
```

This calculates the mean square displacements of all molecules of the requested type, based on the difference between their current positions and their positions when the command was first invoked.

```
int insideecoli(double *pos,double *ofst,double rad,double length);
```

This is a short utility routine used by the command `cmdincludeecoli`. It returns a 1 if a molecule is inside an *E. coli* shape and a 0 if not. `pos` is the molecule position, `ofst` is the physical location of the cell membrane at the center of the low end of the cell (the cell is assumed to have its long axis along the *x*-axis), `rad` is the cell radius used for both the cylindrical body and the hemispherical ends, and `length` is the total cell length, including both hemispherical ends.

```
void putinecoli(double *pos,double *ofst,double rad,double length);
```

This is another short utility routine used by the command `cmdincludeecoli`. It moves a molecule from its initial position in `pos` to the nearest surface of an *E. coli* shape. Parameters are the same as those for `insideecoli`.

```
int molinpanels(simptr sim,int ll,int m,int s,char pshape);
```

This function, which might be better off in the main `smollib.c` code, is used to test if molecule number `m` of live list `ll` is inside any of the `pshape` panels of surface number `s`. Only spheres are allowed currently as panel shapes, because neither rectangles nor triangles can contain molecules. If `s` is sent in with a value less than 0, this means that all `pshape` panels of all surfaces will be checked.

4.3 How to add a new runtime command

Command interpreter routines are designed to be quite modular, so they can be written and removed easily. They are easy to write in some cases, while in other cases one needs to know the intricacies of the data structures in order to properly navigate or modify them. Commands are allowed to look at or modify any part of the simulation structure, making them quite powerful, but also problematic if they are written incorrectly. Thus, proofread and check your commands before releasing them!

To write a command, do the following steps, which can be done in absolutely any order:

1. In the list of runtime commands in the user portion of the documentation, write a description of the new command. It will need a name, which will be listed below as *name*.
2. In `smolib2.c`, add a new declaration to the top of the file for the command, which looks like:

```
int cmdname(simptr sim,cmdptr cmd,char *line2);
```

3. The first function of `smolib2.c` is `docommand`. In it, add an “`else if()`” line for the new command.
4. Write the function for the new command, which follows the declaration listed above.
5. Proofread the function and test the command.
6. Add documentation about the command to the code portion of the documentation. Also add it to the list of code modifications in the final portion of the code documentation.

5. Functions in `smoldyn.c` (simulation control and graphics)

The source code file `smoldyn.c` contains high level functions that allow program entry from the shell, exit to the shell, functions that manage the simulation, and functions that take care of graphics. These functions are all declared locally and thus cannot be called from externally. The structure of this segment is largely determined by the constraints of the OpenGL framework, in which control is passed from the main program to OpenGL and is never returned. As a result, the only way to quit a simulation that uses graphics is either by having the user choose quit from the menu or with the standard library command `exit(0)`. The former method was chosen, but has the drawback that there is no way to free the simulation structure before terminating the program. Instead, *Smoldyn* relies on the system to free the allocated memory. Without graphics, a more conventional C structure is used, including freeing of memory upon completion and a normal return to the shell, although the structure of the code is still slightly strange due to its need to be compatible with the OpenGL segment.

5.1 Non-OpenGL functions

```
int simulatetimestep(simptr sim,int ctr[]);
void endsimulate(simptr sim,int vb,int ctr[],time_t tsst,int er);
void smolsimulate(simptr sim,int vb);
```

```
int main(int argc, char *argv[]);
```

`int simulatetimestep(simptr sim, int ctr[]);`
`simulatetimestep` runs the simulation over one time step. If an error is encountered at any step, or a command tells the simulation to stop, or the simulation time becomes greater than or equal to the requested maximum time, the function returns an error code to indicate that the simulation should stop; otherwise it returns 0 to indicate that the simulation should continue. Error codes are 1 for simulation completed normally, 2 for error with `assignmolecs`, 3 for error with `zeroreact`, 4 for error with `unireact`, 5 for error with `bireact`, 6 for error with `molsort`, or 7 for terminate instruction from `docommand` (e.g. stop command). Errors 2 and 6 arise from insufficient memory when boxes were being expanded and errors 3, 4, and 5 arise from too few molecules being allocated initially.

```
void endsimulate(simptr sim, int vb, int ctr[], time_t tstrt, int er);
```

`endsimulate` takes care of things that should happen when the simulation is complete. This includes executing any commands that are supposed to happen after the simulation, displaying numbers of simulation events that occurred, and calculating the execution time. `er` is a code to tell why the simulation is ending, which has the same values as those returned by `simulatetimestep`. If graphics are used, this routine just returns to where it was called from (which is `TimerFunction`); otherwise, it frees the simulation structure and then returns (to `smolsimulate` and then `main`).

```
void smolsimulate(simptr sim, int vb);
```

`smolsimulate` runs the simulation without graphics. It does essentially nothing other than running `simulatetimestep` until the simulation terminates. At the end, it calls `endsimulate` and returns.

```
int main(int argc, char *argv[]);
```

`main` is a simple routine that provides an entry point to the program. It checks the command line arguments, prints a greeting, inputs the configuration file name from the user, and then calls `setupstructs` to load the configuration file and set up all the structures. If all goes well, it calls `simulate` or `simulategl` to run the simulation.

5.2 Functions that require OpenGL

```
void RenderSurfaces(simptr sim);
void RenderScene(void);
void TimerFunction(int value);
void smolsimulategl(simptr sim, int vb);
```

`void RenderSurfaces(simptr sim);`
 Draws all surfaces in the simulation using OpenGL graphics.

```
void RenderScene(void);
```

RenderScene is the call-back function for OpenGL that displays the graphics. This function simply draws a box for the simulation volume, as well as points for each molecule.

```
void TimerFunction(int er);
```

TimerFunction is the call-back function for OpenGL that runs the simulation. *er* is positive if the simulation should quit due to a simulation error or normal ending, *er* is negative if the simulation has been over, and *er* is 0 if the simulation is proceeding normally. This also looks at the state defined in my *opengl2* library; if it is 0, the simulation is continuing, if it is 1, the simulation is in pause mode, and if it is 2, the user told the simulation to quit. This function runs one simulation time step, posts graphics redisplay flags, and saves TIFF files as appropriate.

```
void smolsimulategl(simptr sim,int vb);
```

smolsimulategl initiates the simulation using OpenGL graphics. It does all OpenGL initializations, registers OpenGL call-back functions, sets the global variables to their proper values, and then hands control over to OpenGL. This function returns as the program quits. *sim* is a pointer to the simulation structure and *vb* is 1 for verbose operation and 0 for non-verbose.

6. *Smoldyn* modifications

6.1 Modifications made for version 1.5 (released 7/03).

Added heirarchical configuration file name support.

Zeroreact assigns the correct box for new molecules.

The user can choose the level of detail for the bimolecular interactions (just local, nearest neighbor, all neighbor, including periodic, etc.)

Bimolecular reactions were slow if most boxes are empty. Solution was to go down molecule list rather than box list.

Absorbing wall probabilities were made correct to yield accurate absorption dynamics at walls.

Cleaned up and got rid of old commands.

The current time input was made useful.

Graphics were improved by adding perspective and better user manipulation.

Simulation pausing was made possible using graphics and improved without graphics.

If a command was used with a wrong file name, the command string became corrupted during the final command call. This was fixed by Steve Lay.

Fixed the neighbor list for bimolecular reactions between mobile and immobile reactants.

Reactions were made possible around periodic boundaries.

Molecules were lost sometimes. This bug was fixed: 4 lines before end of *molsort*:

```
was: while(!live[m]) {  
now: while(!live[m]&& m < nl[11]) {
```

Output files now allow the configuration file to be in a different folder as *Smoldyn*.

Added an output file root parameter.

Added the command `replacexyzmol`. Afterwards, the code for the command was sped up considerably.
Sped up the command `excludebox`.
Command time reports were fixed for type `b` and `a` commands.
Added more types of command timing codes.
Improved accuracy of `unireact` so that it correctly accounts for multiple reactions from one identity.
Improved product parameter entry and calculation, as well as the output about reaction parameters.
Added the routine `checkparams` to check that the simulation parameters are reasonable.

6.2 Modifications made for version 1.51 (released 9/5/03).

Fixed a minor bug in `doreact` which allowed the molecule superstructure indicies to become illegal if not enough molecules were allocated.
Fixed a minor bug in `cmdreact1` which did not check for errors from `doreact`.
Added command `molpos`.
Moved version number from a `printf` statement to a macro, in `smoldyn.c` file.
Added command `listmols2`, from a file sent to me by Karen Lipkow.
Fixed a minor bug in `checkparams` that printed warnings for unused reactions.
In `simulatetimestep` in `smoldyn.c`, the order of operations was `diffuse`, `checkwalls`, and then `assignmolecs`. The latter two were swapped, which should make wall checking more accurate when time steps are used that are so long that rms step lengths are a large fraction of box sizes. The new version is less accurate than before when the simulation accuracy is less than 10, but should be more accurate when it is 10.
Replaced the `coinrand` call in `unireact`, which determines if a reaction occurred, with `coinrand30` to allow better accuracy with low probabilities. Also changed the relevant check in `checkparams`.
Improved reactive volume test in `checkparams`.
Increased `RANDTABLEMAX` from 2047 to 4095.
Some modifications were made to `random.h`.
Fixed a major bug in `rxnfree`, regarding the freeing of the table elements.

6.3 Modifications made for version 1.52 (released 10/24/03)

Changed comments in `rxnparam.h` and `rxnparam.c`, but no changes in code.
Changed `cmdsavesim` in `smollib2.c` to allow it to compile with `gcc`.
Added another call to `assignmolecs` in `simulatetimestep` in `smoldyn.c`, after the call to `checkwalls`, to make sure that all molecules are assigned properly before checking reactions. This slows things down some, but should allow slightly longer time steps.

To the `opengl2.c` file, the `KeyPush` function was modified so now pressing 'Q' sets the `Gl2PauseState` to 2, to indicate that a program should quit. A few modifications were also made in `smoldyn.c` function `TimerFunction` to make use of this.

Corrected two significant bugs in the `checkwalls` function in `smollib.c` regarding absorbing walls. First, it didn't work properly for low side walls. Also, the probability equation was incorrect, which was noticed by Dan Gillespie.

Fixed a minor bug in `cmdsavesim` in `smollib2.c` file, which caused an output line for `rate_internal` to be displayed for declared but unused reactions.

Several commented out functions in `loadrxn` were removed because they were obsolete and have been replaced by `product_param`. They were: `p_gem`, `b_rel`, `b_abs`, `offset`, `fixed`, and `irrev`.

A command superstructure was created, which moved several structure elements out of the simulation structure. No new functionality was created, but the code is cleaner now. New routines are `cmdssalloc` and `cmdssfree`. Updated routines are: `simalloc`, `simfree`, `loadsimul`, `setupstructs`, `cmdoutput` (including function declaration), `openoutputfiles` (including function declaration and ending state if an error occurs), `commandpop` (including function declaration), `checkcommand`, `endsimulate`, `savesim`, `main`, and all commands that save data to files.

Renamed the "test files" folder to "test_files".

6.4 Modifications made for version 1.53 (released 2/9/04)

Cleaned up commands a little more by writing routine `getfptr` in `smollib2.c` and calling it from commands that save data, rather than repeating the code each time.

All routines that dealt with the command framework were moved to their own library, called `SimCommands`. This also involved a few function name and argument changes, affecting `smoldyn.c`, `smollib.c`, `smollib.h`, `smollib2.c`, and `smollib2.h`.

Formatting was cleaned up for structure output routines.

Swapped drawing of box and molecules, so box is on top. Also increased default box line width to 2 point.

Computer now beeps when simulation is complete.

Modified `SimCommand` library so that each invocation of a command is counted and also changed declaration for `docommand` in `smollib2`. This change was useful for improving the command `listmols2` so it can be run with several independent time counters. Also, wrote command `listmols3`.

Wrote the new configuration file statement `boxsize`.

Wrote the new commands `excludesphere` and `includeecoli`.

Wrote the commands `overwrite` and `incrementfile`, which also involved some changes to the `SimCommand` library and required the new configuration file statement `output_file_number`.

Added a new configuration file statement `frame_thickness`.

When simulation is paused using OpenGL, the simulation time at which it was paused is now displayed to the text window.

6.5 Modifications made for version 1.54 (released 3/3/04)

Swapped order of commands and OpenGL drawing so that commands are executed before displaying results. Also wrote section 3.2 of the documentation to discuss this ordering and other timing issues.

Wrote documentation section 3.3 on surface effects on reaction rates and added the *reactW* set of test files.

6.6 Modifications made for version 1.55 (released 8/20/04)

Improved graphics manipulations and added ability to save image as a TIFF file. This is not documented yet.

Made a few tiny changes in random.c and string2.h and .c.

The configuration file statement `max_cmd` is now obsolete because the command queue is automatically created and expanded as needed. Also, lots of changes were made to the library file `SimCommand.c` so that there are now two command queues: one is as before and uses floating point times for command execution and the other uses an integer counter for commands that are supposed to happen every, or every n 'th, iteration.

Added error strings to commands as well as the macro statement `SCMDCHECK`.

6.7 Modifications made for version 1.56 (released 1/14/05)

Made lots of changes in `opengl2.c`.

`#include` files for `gl.h` and `glut.h` now use brackets rather than quotes.

Improved graphics significantly.

Rewrote `TimerFunction` to clarify code.

Added ability to save TIFF stacks which can be compiled into movies.

Added keypress command.

Added comments to the code.

User and programmer parts of documentation were split to separate files.

6.8 Modifications for version 1.57 (released 2/17/05)

Added command `setrateint`.

6.9 Modifications for version 1.58 (released 7/22/05)

Fixed 2-D graphics so they a border is now shown again around the simulation volume.

Added runtime commands `replacevolmol` and `volumesource`.

Random number table for diffusion is now shuffled before use, which significantly reduces errors from an imperfect random number generator.

Added position ranges to `mol` command.

6.10 Modifications for version 1.59 (released 8/26/05)

Random number seed is now stored and is displayed before a simulation starts.

6.11 Modifications for version 1.60 (not released, but given to Karen 9/30/05)

Fixed a small bug in checkparams.

6.12 Modifications for version 1.70 (released 5/17/06)

Added reflective, absorbing, and transparent surfaces for 1 to 3 dimensions with panel shapes that can be: rectangle, triangle, and sphere.

Geometry.c and its header Geometry.h are new libraries that are used.

Added background and frame color options.

Reformatted and significantly updated part 2 of the Smoldyn documentation. Added surface descriptions to part 1 of documentation.

Changed molecule sorting in molsort so that list compacting maintains list order.

Wrote reassignmolecules to replace assignmolecules, which should increase efficiency and allow accurate surface treatment.

Made it possible to load molecule names individually rather than all at once. New configuration file statements are max_name and name.

Added pointers to the live molecule lists called topl (and renamed top to topl), which will differentiate old molecules from the new “reborn” ones. This is important for treating surfaces after reactions.

6.13 Modifications for version 1.71 (released 12/8/06)

Added glutInit call to main function in smoldyn.c.

Changed OpenGL drawing slightly for surfaces, so now 3D surface colors are always the same on the front and back, but can also be semi-transparent, although with OpenGL errors.

Added command killmolinsphere.

Cleaned up simulation loading some, with minor modifications in setupstructs, loadsimul, and setupboxes, as well as writing of setdiffusion. This makes it so that molecule sorting only happens in molsort, and it took some unwanted code out of loadsimul.

Added molecule serial numbers to the molecule structure and superstructure.

Added some elements to command structures so that commands now have storage space.

Added RnSort.c library to project, as well as some new functions in RnSort.c.

Added command meansqrdisp.

Completely rearranged order of functions in smolllib.c and in documentation part II.

Cleaned up surface code. Fixed rendering of 3-D spheres. Added support for cylinders and hemispheres.

Added statements: grid_thickness and block comments with /* and */.

To action_front and similar statements, allowed “all” for molecule name.

Tried to stop diffusing molecules from leaking across reflective surfaces.

6.14 Modifications for version 1.72 (released 2/26/07)

Finally got reflective surfaces to stop leaking diffusing molecules. This involved many changes in the surface code sections.

Walls are no longer functional when any surfaces are defined, so new surfaces have to be defined to serve as system boundaries. Also, periodic surfaces are now possible.

Changed all float data types to doubles throughout smollib.c, smollib2.c, Geometry.c, smoldyn.c, and their headers.

Made it so that bimolecular reactions across surfaces can only happen with transparent surfaces.

Two dimensional graphics now allow panning and zooming.

7. The wish list/ to do list

7.1 Bugs to fix

3-D box cross testing, done in Geometry.c, isn't correct for all shapes (triangles in particular, maybe others).

Reactions can happen across periodic boundaries, as they should, but potential obstructing surfaces are not investigated in these cases. This may be too much work to fix, but should at least be documented.

The savesim command needs to be updated significantly.

7.2 Desired features

More internal surface support. There is a tremendous amount that would be very nice here. Simple reflective surfaces have finally been added. Next are semi-permeable surfaces, non-diffusing membrane-bound molecules, diffusing membrane-bound molecules, moveable surfaces, etc.

Molecules with excluded volume. The idea is to define "non-reactive reactions", in which a molecule collision would be declared exactly as they are currently for reactions, but the result would not be a reaction, but a return of both molecules to their starting points. This probably requires an asynchronous design such that molecules are diffused and interacted one at a time.

Other non-reactive interactions, such as allostery.

Complex formation. A complex could be allowed for multiple molecules that are similar or different molecules, with a K_D . While complexed, they would diffuse together, and go about their reactions in their normal way. Allowing complexes would reduce combinatorial explosion problems that occur when each one has to be declared as a separate species.

Fibers (such as DNA), fiber-bound molecules, etc. Also, membrane-bound polymers would be nice.

Variable length simulation time steps. The less difficult method would be to figure out how fast the system is changing as a whole and to adjust the simulation time step to compensate for this. A challenge though, is that it is nearly impossible to redo a step that was determined to have been too long without introducing significant statistical bias. The more difficult method is to use different length time steps for different molecules, so as to poll labile ones more often than stable ones.

More functionality for the runtime command interpreter. It would be nice if commands could communicate with each other, have their own storage space (done for v. 1.71), etc. An idea for this is to establish a bulletin board within the command superstructure, on which commands could post and read memos. More generally, this could be expanded into an entire programming language if desired, although it would take some thought on how to do it in the best way.

A new runtime command for more versatile text output. Rather than having a pile of specialized output commands, it would be nice to have something akin to a print statement, where any of a wide variety of simulation variables could be printed with a user-defined format.

Easier compilation for different systems, including pre-compiled code, Make files where needed, a better collection of stuff in the download folder, advice on OpenGL code access and configuration, compiling advice, etc.

Configuration file compatibility with SBML, XML, or other standards.

Addition of a graphical user interface, or at least a better user interface. For example, it would be nice to be able to browse the available configuration files using a standard open file window, rather than having to know the exact path and filename.

Inclusion of continuous concentrations for chemical species that are abundant. Ideally, these concentrations should be updated with ODEs, PDEs, spatial- or non-spatial Langevin dynamics, or spatial- or non-spatial Gillespie algorithm, according to the user's choice.

Modify main to include flags on input line.

Better graphics, including lighting sources and translucent molecules. POV-ray may be a good graphics program to work with.

The function assignmolecs should be deleted once it has been confirmed that reassnmolecs is working properly.

The load functions should be cleaned up, including loadsimul, loadrxn, and loadsurface. The filestack stuff should be encapsulated in its own set of functions, which would also automatically take care of several commands, such as comments, end_file, and read_file.